下载APP

25｜可执行二进制文件里有什么？

2022-02-18 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 14:51 大小 13.60M

你好，我是于航。从这一讲开始，我们就进入到了“C 程序运行原理篇”的学习。

和之前的内容相比，在这一模块中，我们将会从“台前”走向“幕后”：从由 C 代码直观表示的程序逻辑，走向程序在运行过程中，背后与操作系统交互时的具体原理。相信学习完这个模块后，你会对一个 C 程序从代码编写，到通过编译，再到最终被操作系统运行的完整过程有更深入的理解。其中，程序的运行细节仅与所在操作系统紧密相关，因此一模块中介绍的原理性知识也同样适用于由 Rust、C++，以及 Go 等其他系统级编程语言编写的程序。

领资料

而今天我们先来看下，经常被提及的“二进制可执行文件”究竟是什么。

可执行文件的格式

我们都知道，一份 C 代码在经过编译器处理后，便可得到能够直接运行的二进制可执行程序。而在不同操作系统上，这些编译生成的可执行文件都有着不同的特征，其中最明显的差别便是文件后缀名。比如，在 Microsoft Windows 操作系统上，通常会以 “.exe” 为后缀名来标注可执行文件；而在类 Unix 操作系统上，可执行文件通常没有任何后缀名。

除此之外，更重要的不同点体现在各类可执行文件在内部数据的组织和结构上。通常来说，最常见的几种可执行文件格式有针对微软 Windows 平台的 PE（Portable Executable）格式、针对类 Unix 平台的 ELF（Executable and Linkable Format）格式，以及针对 MacOS 和 IOS 平台的 Mach-O 格式。

另外，值得一提的是，在 Unix 系统诞生早期，那时的可执行程序还在使用一种名为 “a.out” 的可执行文件格式。“a.out” 的全称为 “Assembler Output”，直译过来即“汇编器输出”。该名称来源于 Unix 系统作者 Ken Thompson 最早为 PDP-7 微型计算机编写的汇编器的默认输出文件名。时至今日，这个名称依然是某些编译器（比如 GCC）在创建可执行文件时的默认文件名。不仅如此，作为第一代可执行程序格式，它对后续出现的 ELF、PE 等格式也有着重要的参考意义。

接下来，我就以类 Unix 平台上最常使用的 ELF 格式为例，来带你看看这些可执行文件格式，究竟是以怎样的方式存储应用程序数据的。

ELF 文件格式

不同的可执行文件格式会采用不同方式，来组织应用程序运行时需要的元数据。但总体来看，它们对数据的基本组织方式都符合这样一个特征：**使用统一的“头部（header）”来保存可执行文件的基本信息。而其他数据则按照功能被划分在了以 Section 或 Segment 形式组织的一系列单元中**。当然，ELF 格式也不例外。

> 需要注意的是，在一些中文书籍和文章中，Section 和 Segment 这两个单词可能会被统一翻译为“段”或“节”。但对于某些格式，比如 ELF 来说，它们实际上则分别对应着不同的概念，因此为了保证同学们理解的准确性，这里我直接保留了英文。

接下来，让我们从一个真实的 C 程序入手，通过观察这个程序对应二进制文件的内容，你可以得到对 ELF 格式基本结构的一个初步印象。该程序的 C 源代码如下所示：

复制代码

```c
// elf.c
#include <stdio.h>
int main (void) {
  const char* str = "Hello, world!";
  printf("%s", str);
  return 0;
}
```

经过编译后，我们可以得到上述代码对应的二进制可执行文件。接下来，使用 file 命令可以确认该文件的格式信息。该命令的执行返回结果如下图所示：

根据命令执行结果开头处的信息，我们可以确认这是一个 ELF 格式的可执行文件。其中的 64-bit 表示该文件采用的是 64 位地址空间。除此之外，命令还回显出了该 ELF 格式的版本，是否采用动态链接，以及使用的动态链接器地址等信息。

接下来，我们通过 readelf 命令来查看该可执行文件的内部组成结构。顾名思义，这个命令专门用于读取特定 ELF 格式文件的相关信息。

ELF 头

通过为 readelf 指定 “-h” 参数，我们可以观察该文件的 ELF 头部内容。命令执行结果如下图所示：

ELF 头内包含有描述整个可执行文件重要属性的相关信息。应用程序在被执行时，操作系统可以借助其头部的相关字段，来快速找到支持程序运行所需要的数据。

其中，操作系统通过 Magic 字段来判断该文件是不是一个标准的 ELF 格式文件，该字段一共长 16 个字节，每个字节代表着不同含义。前四个字节构成了 ELF 文件格式的“魔数”，第一个字节为数字 0x7f，后三个字节则对应于三个大写字母 “ELF” 的 ASCII 编码。剩下的字节还标记出了当前 ELF 文件的位数（如 32/64）、字节序、版本号，以及 ABI 等信息。

除该字段外，ELF 头中还包含有 ELF 文件类型、程序的入口加载地址（0x4004b0），即程序运行时将会执行的第一条指令的位置，以及该可执行文件适用的目标硬件平台和目标操作系统类型等信息。ELF 作为一种文件格式，不仅在可执行文件中被使用，静态链接库、动态链接库，以及核心转储文件等也都可以采用这种格式。我们会在下面的 “ELF 文件类型” 小节中继续讨论这个问题。

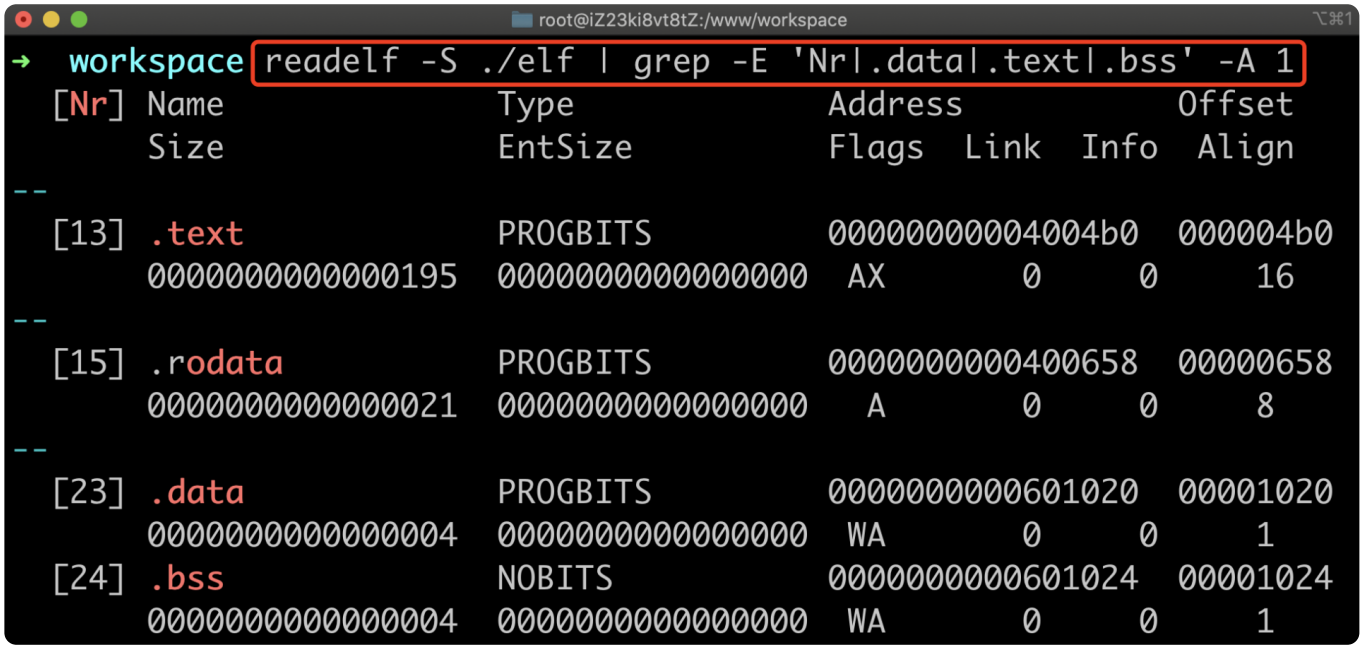
ELF Section 头

在 ELF 格式中，Section 用于存放可执行文件中按照功能分类好的数据，而为了便于操作系统查找和使用这些数据，ELF 将各个 Section 的相关信息都整理在了其各自对应的 Section 头部中，众多连续的 Section 头便组成了 Section 头表。

Section 头表中记录了各个 Section 结构的一些基本信息，例如 Section 的名称、长度、它在可执行文件中的偏移位置，以及具有的读写权限等。而操作系统在实际使用时，便可直接从 ELF 头部中获取到 Section 头表在整个二进制文件内的偏移位置，以及该表的大小。

通过观察上图中的 ELF 头信息，我们能够得知，该 ELF 文件内包含有 30 个 Section 头，即对应 30 个 Section 结构，且第一个 Section 头位于文件开始偏移第 15512 个字节处。而通过为 `readelf` 命令指定 “-S” 参数，我们可以查看所有这些 Section 头的具体信息。

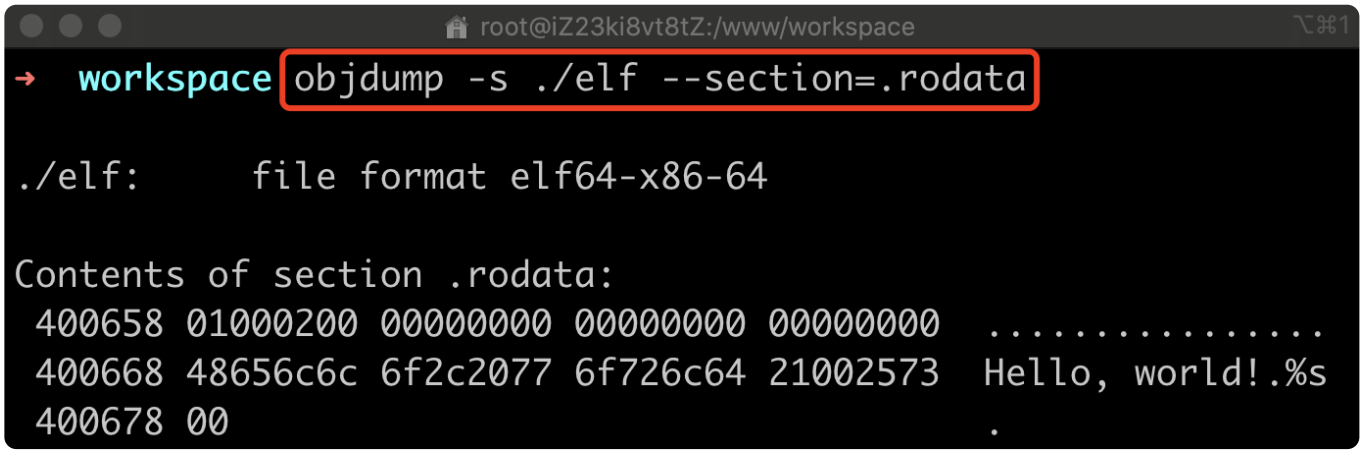
该命令的执行结果如下图所示（这里限于篇幅，我只列出了较为重要的几个 Section 头部的内容）：



可以看到，这里我主要筛选出了 .text、.rodata、.data、.bss 这四个 Section 对应头部的详细内容。如果你还记得我在 02 讲 和 10 讲 中介绍过的数据存储位置的相关知识，那对这四个 Section 一定不会陌生。其中，.text 主要用于存放程序对应的机器代码；.rodata 用于存放程序中使用到的只读常量值；.data 中包含有程序内已经初始化的全局变量或静态变量的值；而 .bss 中则存放有初始值为 0 的全局或静态变量值。

Section 头部中也标记了各个 Section 实际数据的所在位置。对于 .rodata 来说，我们可以在文件偏移第 0x658 个字节，或程序运行时在进程 VAS 中的偏移位置 0x400658 处看到它的实际内容。这里我们可以用 objdump 命令来验证一下。

objdump 命令是一个可以用来查看二进制文件内容的工具，通过为它指定 “-s” 参数，我们可以查看某个 Section 的完整内容。该命令的执行结果如下所示：

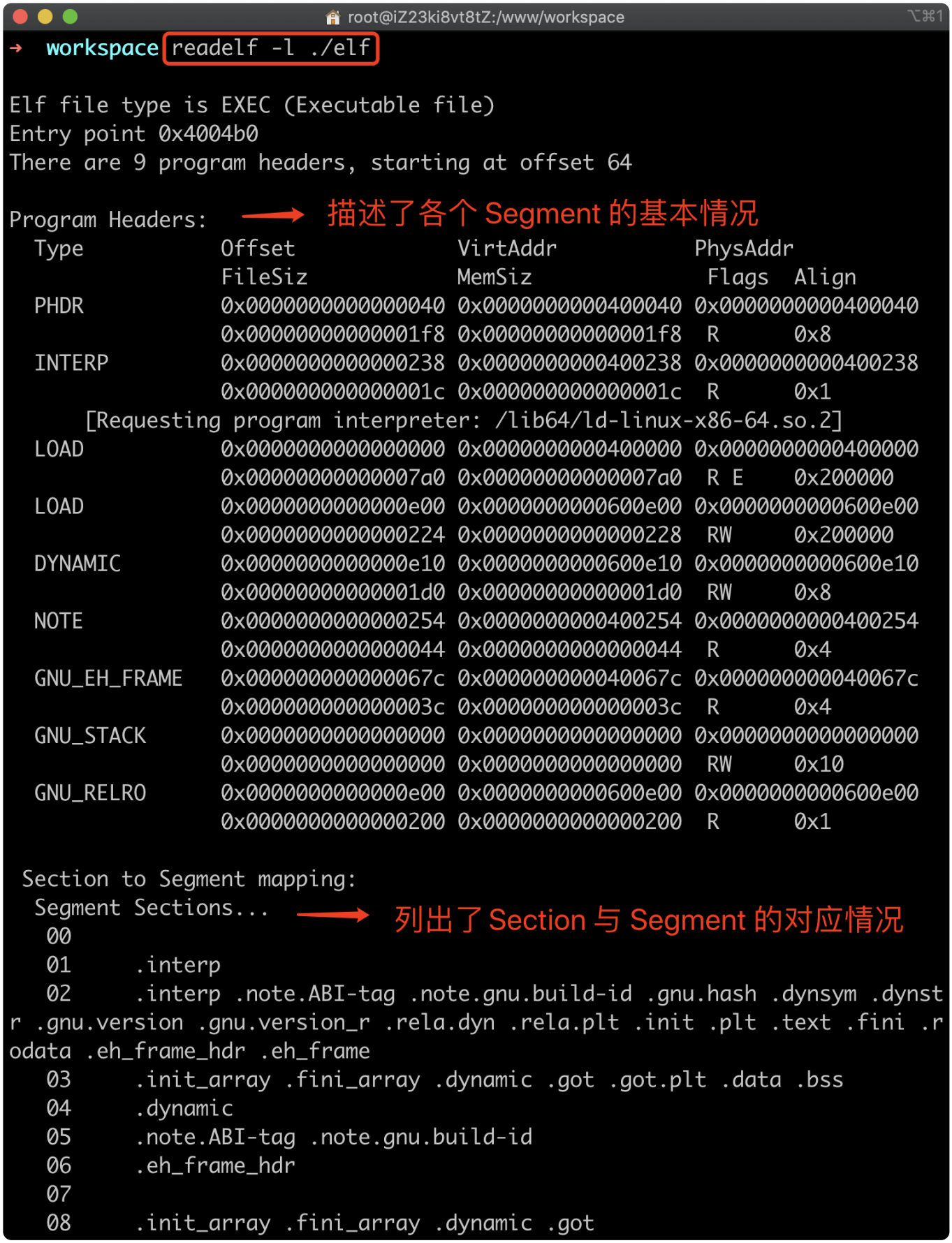


可见，我们在 C 代码中使用到的字符串数据 “Hello, world!” ，便被放置在了该 Section 距离其开头偏移 0x10 字节的位置上。

在 ELF 格式中，众多的 Section 组成了描述该 ELF 文件内容的静态视图。而静态视图的一大作用，便是完成应用程序整个生命周期中的“链接”过程。链接意味着不同类型的 ELF 格式文件之间会相互整合，并最终生成可执行文件，且该文件可以正常运行的过程。根据整合发生的时期，链接可以被分为“静态链接”与“动态链接” ，这部分内容我会在后面的 27 讲与 29 讲中再为你深入介绍。

ELF Program 头

除了由 Section 组成的静态视图外，众多的 Segment 则组成了描述可执行文件的动态视图。Segment 指定了应用程序在实际运行时，应该如何在进程的 VAS 内部组织数据。同样地，我们也可以通过为 readelf 命令指定 “-l” 参数，来观察这一讲开头那个程序对应可执行文件的 Segment 情况。该命令的执行结果如下图所示：



与 Section 类似的是，每个 Segment 也都有其对应的头部，以描述该 Segment 的一些基本信息，我们一般将其称为 Program 头。

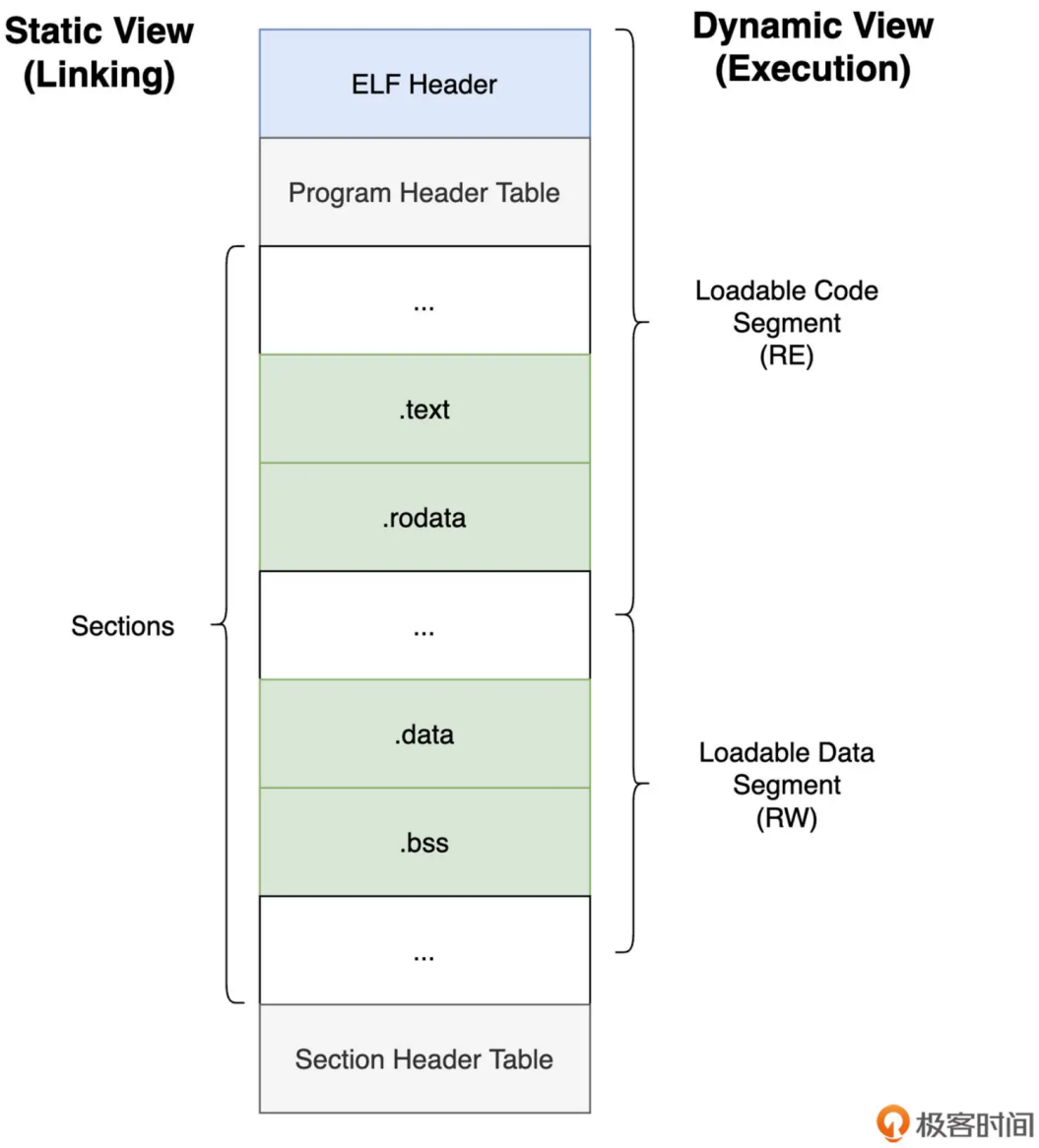
Program 头中包含着各个 Segment 的类型、偏移地址、大小、对齐情况，以及权限等信息。其中，被标注为 “LOAD” 类型的 Segment 将会在程序运行时被真正载入到进程的 VAS 中，而其余 Segment 则主要用于辅助程序的正常运行（比如进行动态链接）。不仅如此，Program 头表的具体偏移位置和大小也被放置在了 ELF 头部中，因此操作系统可以在需要时，随时快速地得到这些信息。

通常来说，各个 Segment 与 Section 之间会有一定的对应关系。比如在上面的图片中，第一个 LOAD 类型的 Segment 便包含有 .text 和 .rodata 在内的多个 Section，而 .data 则被包含在第二个 LOAD 类型的 Segment 中。如果进一步观察，你会发现，第一个 LOAD Segment 具有的权限为 “RE” ，也就是可读可执行；而第二个 LOAD Segment 具有的权限为 “RW” ，即可读可写。那为什么这样分配呢？相信此时你一定有了答案。

另外我们观察到，第一个 LOAD Segment 所包含的内容，对应到可执行文件内的偏移（Offset）为 0。这意味着，操作系统在执行该程序时，除了各个 Segment 对应的 Sections 外，它还会将该文件的 ELF 头，连同它的 Program 头表一同加载到内存中。

到这里，我们已经对 ELF 格式二进制可执行文件的内部情况有了一个大致了解，如果想了解更多有关 ELF 格式的详细信息，你可以参考这个链接。

虽然上面的内容没有涉及 ELF 内部的所有设计细节，但对于日常学习而言，其实掌握 ELF 格式的基本组成结构（ELF 头、Section 与 Segment 分别对应的静态视图和动态视图）就足够了。你可以参考下图，来更加直观地回顾一下这些内容：



接下来，我们再一起看看如何使用 C 语言进行 ELF 编程。

ELF 编程

目前在 Linux 系统中，我们可以直接使用内核提供的头文件 elf.h 来进行针对 ELF 格式的应用编程。在该头文件中，预定义有针对不同 ELF 概念实体的各类结构类型与宏。

比如，对于 ELF 头部，我们可以直接在代码中使用该头文件中定义的 ElfN_Ehdr（N 根据所在系统的不同，可能取 32 或 64）类型来表示。下面这段 C 代码展示了如何使用这个类型：

复制代码

```
1 #include <stdio.h>
2 #include <elf.h>
3
```

```
4  void print_elf_type(uint16_t type_enum) {
5    switch (type_enum) {
6      case ET_REL: printf("A relocatable file."); break;
7      case ET_DYN: printf("A shared object file."); break;
8      case ET_NONE: printf("An unknown type."); break;
9      case ET_EXEC: printf("An executable file."); break;
10     case ET_CORE: printf("A core file."); break;
11   }
12 }
13
14 int main (void) {
15   Elf64_Ehdr elf_header;
16   FILE* fp = fopen("./elf", "r");
17   fread(&elf_header, sizeof(Elf64_Ehdr), 1, fp);
18   print_elf_type(elf_header.e_type);  // "An executable file."
19   fclose(fp);
20   return 0;
21 }
```

在这段 C 代码中，我们简单地打开了当前目录下名为 “elf” 的二进制可执行文件，并从它的开头处直接读取了对应 Elf64_Ehdr 类型大小的数据，存放到名为 elf_header 的变量中。最后，通过访问该结构对象的 e_type 字段，我们便可得到该 ELF 文件的类型。在 elf.h 头文件中，定义有众多表示 ELF 特定指标的宏常量，比如，若 e_type 字段的值等于宏常量 ET_EXEC，那么表示该文件是一个可执行文件。

可以说，**elf.h 头文件中包含有可用于描述所有合法 ELF 格式文件的各种自定义类型**。因此，通过选择性地阅读和实践，来深入了解 ELF 的设计细节不失为一个好方法。如果想了解关于该头文件的更多信息，你可以通过命令 “man 5 elf” 来查看有关该头文件的 Linux 帮助文档，或者直接在这里找到它的在线版本。

ELF 文件类型

在这一讲结束之前，让我们再来看一个问题： ELF 作为一种文件格式，它究竟在被哪些类型的文件使用呢？

通过上一小节的编程实战，我们可以得知：在 elf.h 头文件的定义中，ELF 格式可以应用在四种不同的文件类型上，它们对应的宏常量分别是 ET_REL、ET_DYN、ET_EXEC，以及 ET_CORE。

这里，我将上述四个宏常量与它们对应的 ELF 文件类型整理在了下面的表格中，供你参考：

宏常量（elf.h）	文件类型	文件定义
ET_RE	可重定位文件	通常指通过链接器的静态链接处理后，能够生成新的可执行文件或共享库文件的一类 ELF 文件类型。常见的 .o 与 .a 文件便属于这一类
ET_DYN	共享目标（共享库）文件	指的是可以在程序运行时，通过动态链接方式使用其功能，并支持程序正常运行的一系列文件。在类 Unix 系统中，它们一般以 .so 结尾
ET_EXEC	可执行文件	可执行应用程序的主要文件类型
ET_CORE	核心转储文件	一般出现在进程意外终止时，操作系统可以将与进程相关的一些信息存放到该文件中，以作调试之用

极客时间

这四种 ELF 文件类型，虽然名称各不相同，但其内部数据的整体组织方式都遵循同样的 ELF 文件格式标准。而不同点在于，由于每种文件类型的功能定位各不相同，因此其内部的 ELF 格式组成结构也各有差异。

就拿可重定位文件来说吧，该类型文件可用于支持大型项目的增量式开发，也就是将程序中可以模块化、独立分发的功能进行单独编译，并形成可重定位文件。而依赖于这些功能实现的应用程序代码，便可与这些可重定位文件一起编译。最后，在经过链接器的静态链接处理后，便能够得到程序对应的可执行文件。这种方式的好处在于，当每次程序功能发生变化时，都可以将需要重新编译的代码约束在最小的范围。

可重定位文件内仅包含有 Section 的相关信息，而没有 Program 头等用于支持其运行的 ELF 结构，因此该类型的文件无法被直接运行。而静态链接的一个主要作用，便是根据程序在 `main` 函数内的调用情况，收集各个可重定位文件中需要使用的功能实现，并最终生成对应含有 Program 头的可执行文件。那么，这个过程具体是怎样进行的呢？我将在下一讲中为你揭晓答案。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲的内容主要是以可执行二进制文件作为切入点的。我首先介绍了在不同操作系统上的几种常见可执行文件格式，然后以最常见的 ELF 格式为例，带你对它的组成细节进行了更为深入的探究。

ELF 文件格式的基本组成结构可以被划分为 ELF 头、Section 和 Segment 三大主要部分。其中，各个 Section 中包含有按照功能类别划分好的、用于支撑 ELF 功能的各类数据。这些数据共同组成了 ELF 文件的静态视图，以用于支持 ELF 文件的链接过程。而众多的 Segment 则组成了 ELF 文件的动态视图，该视图描述了 ELF 文件在被操作系统加载和执行时，其依赖的相关数据在进程 VAS 内的分布情况。

除了通过操作系统自带的 `readelf` 等工具来观察 ELF 文件的内部情况外，我们也可以利用 Linux 内核提供的 elf.h 头文件。该头文件内预先定义了众多的 ELF 元素类型，可辅助我们编写符合自身需求的 ELF 分析和处理工具。

最后，我们还探讨了几种不同 ELF 文件类型之间的区别。可重定位文件、共享目标文件、可执行文件，以及核心转储文件，它们虽然有着各自不同的应用场景和内部数据组成，但也都作为 ELF 文件类型的一种，遵循着 ELF 格式的基本规则。

思考题

尝试编写一个 C 程序，该程序可以读取并打印一个指定 ELF 文件的 Section 信息（Section 名称、大小，以及偏移地址）。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | 实战项目（下）：一个简单的高性能 HTTP Server

下一篇 26 | 进程是如何使用操作系统内存的？

更多学习推荐

精选留言 (2)

写留言

白花风信子
2022-02-19

最期待的就是原理篇了！！！

2

LDxy
2022-02-18

Elf64_Ehdr* elf_header;为指针类型，应该先让其指向内存空间再使用fread

作者回复: 感谢指正！

1